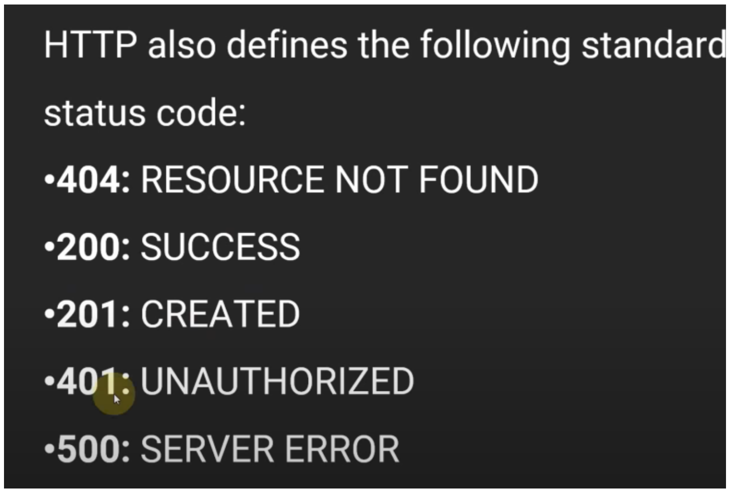

HTTP also defines the following standard status code:

- •404: RESOURCE NOT FOUND
- •200: SUCCESS
- •201: CREATED
- •401: UNAUTHORIZED
- •500: SERVER ERROR

Questions And Answers

Difference between RestController and Controller

Difference between request mapping and get mapping

Can we check environment properties

How to test Spring Boot Application

POM.xml explain

What server spring boot provides

Explain component scan and Bean

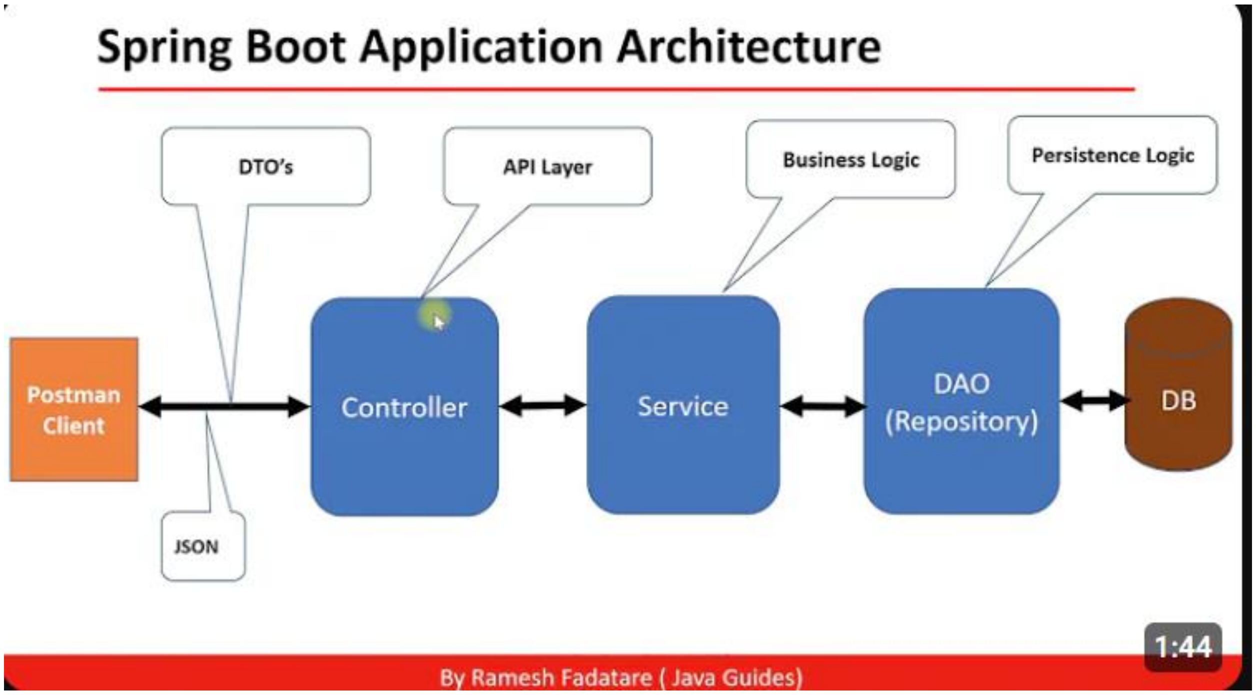
How to Enable debugging log

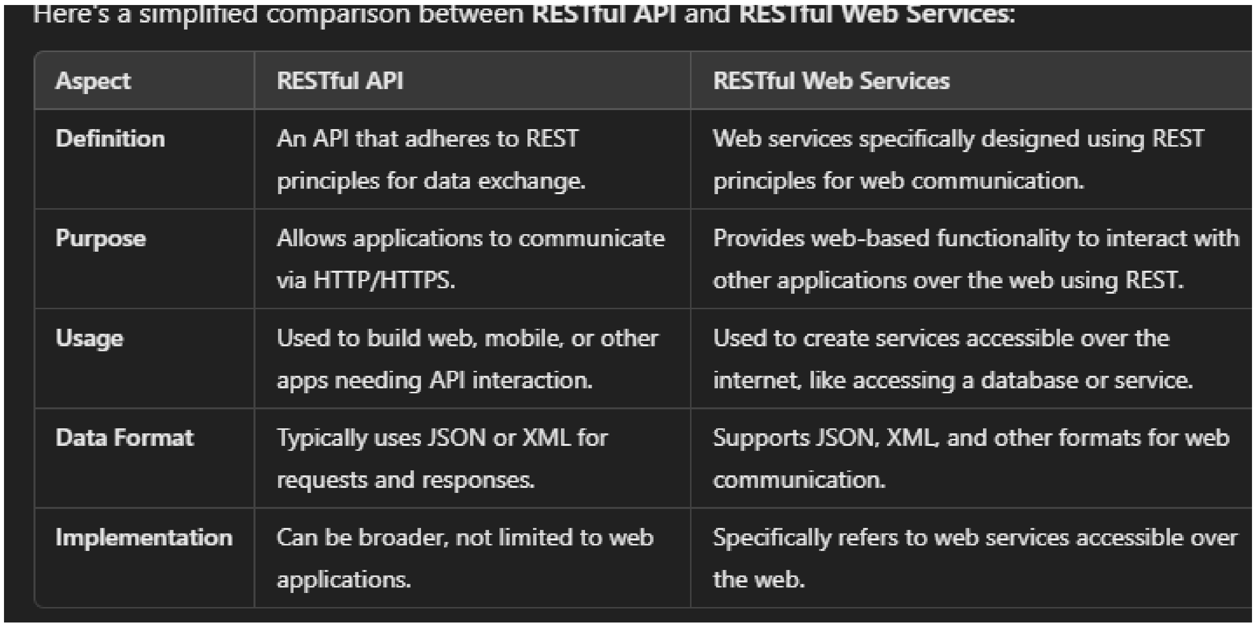
What is purpose of Spring Boot Starter

What are the rest Api for best Practices

Few best practices when you apply collection in project what things you consider

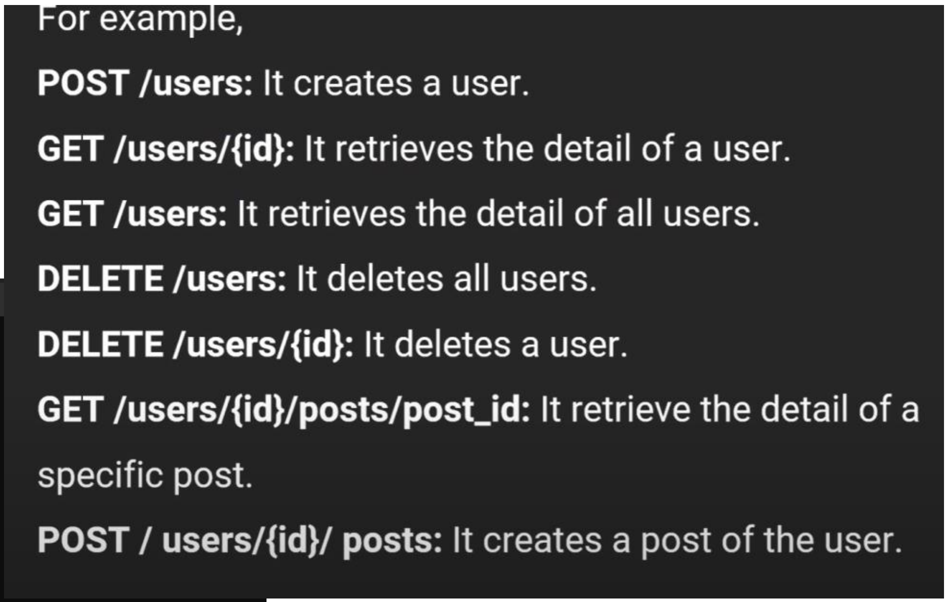
Have you used design pattern in your project





Here's a simplified comparison between **RESTful API** and **RESTful Web Services**:

Aspect	RESTful API	RESTful Web Services
Definition	An API that adheres to REST principles for data exchange.	Web services specifically designed using REST principles for web communication.
Purpose	Allows applications to communicate via HTTP/HTTPS.	Provides web-based functionality to interact with other applications over the web using REST.
Usage	Used to build web, mobile, or other apps needing API interaction.	Used to create services accessible over the internet, like accessing a database or service.
Data Format	Typically uses JSON or XML for requests and responses.	Supports JSON, XML, and other formats for web communication.
Implementation	Can be broader, not limited to web applications.	Specifically refers to web services accessible over the web.



For example,

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

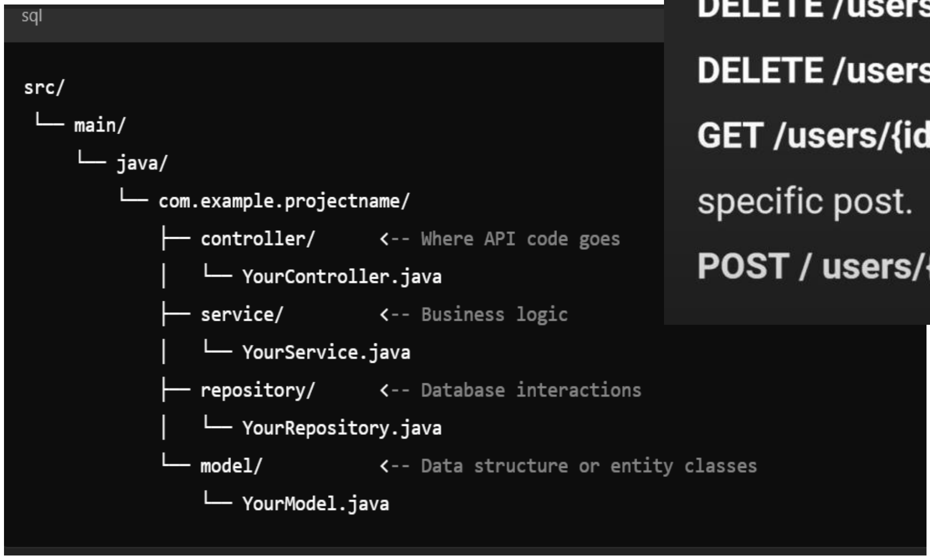
GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

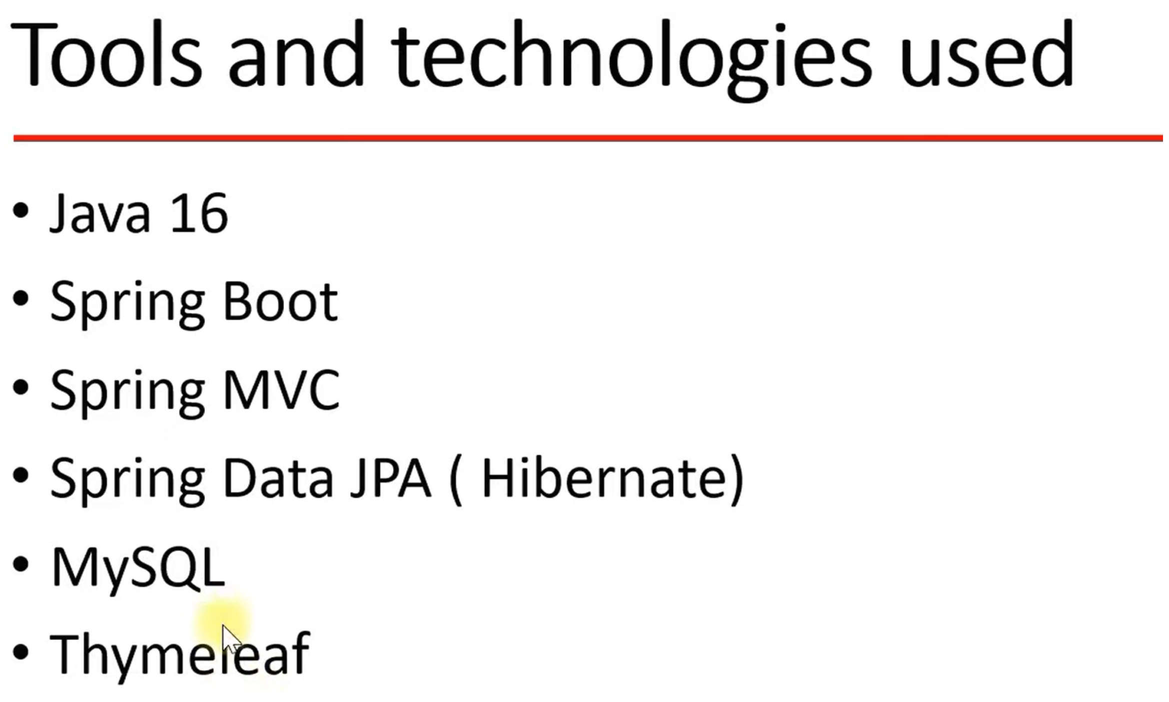
POST / users/{id}/ posts: It creates a post of the user.



sql

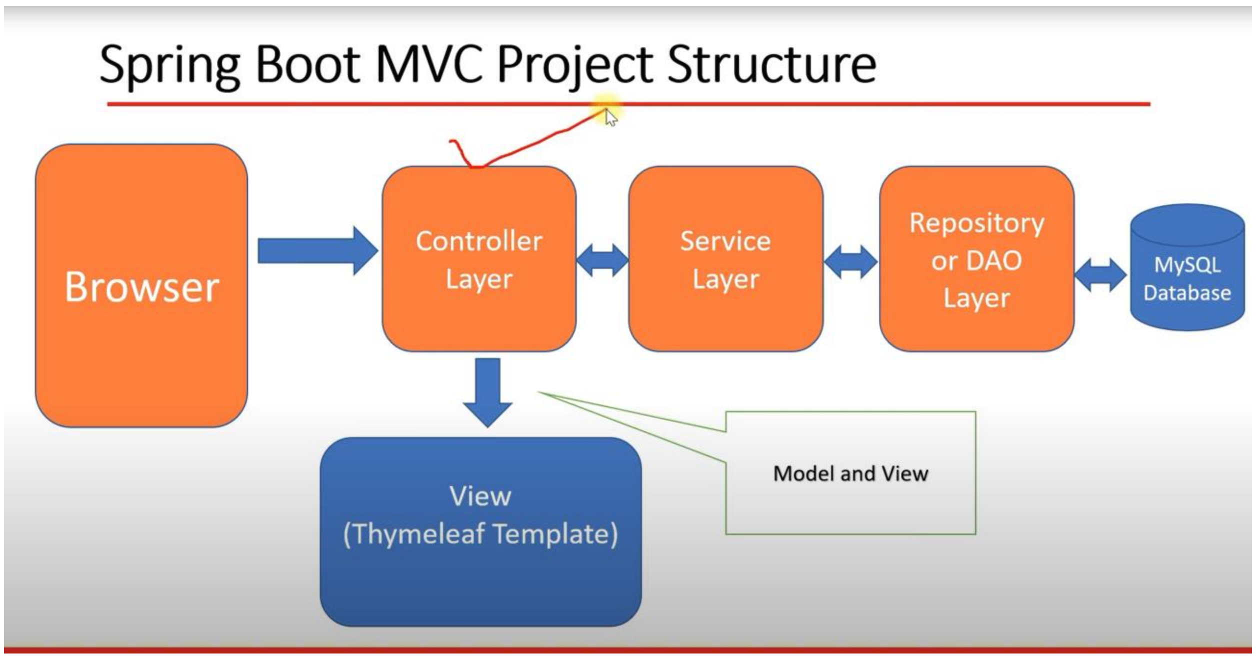
```
src/
 └── main/
     └── java/
         └── com.example.projectname/
             ├── controller/      <-- Where API code goes
             │   └── YourController.java
             ├── service/         <-- Business logic
             │   └── YourService.java
             ├── repository/      <-- Database interactions
             │   └── YourRepository.java
             └── model/           <-- Data structure or entity classes
                 └── YourModel.java
```

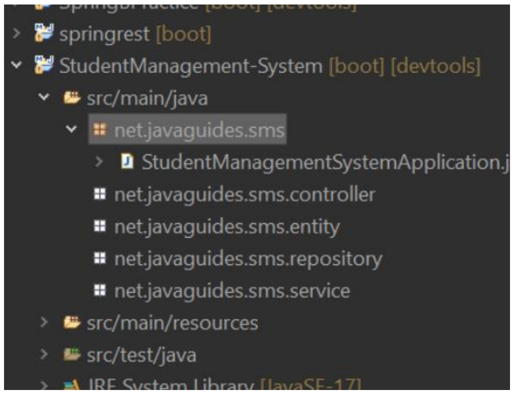
New Project →



Tools and technologies used

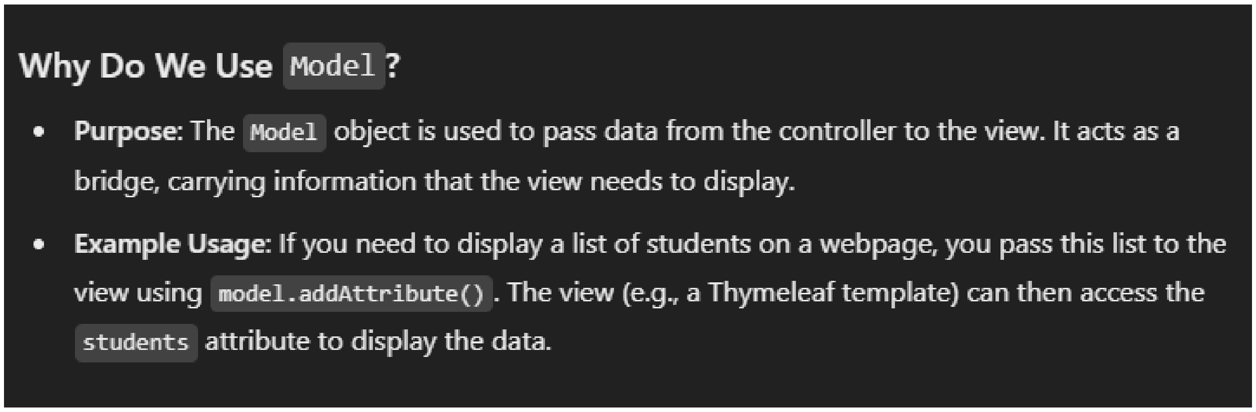
- Java 16
- Spring Boot
- Spring MVC
- Spring Data JPA (Hibernate)
- MySQL
- Thymeleaf





```
> springrest [boot]
v StudentManagement-System [boot] [devtools]
  v src/main/java
    v net.javaguides.sms
      > StudentManagementSystemApplication.j
      net.javaguides.sms.controller
      net.javaguides.sms.entity
      net.javaguides.sms.repository
      net.javaguides.sms.service
  > src/main/resources
  > src/test/java
```

Created all Important packages



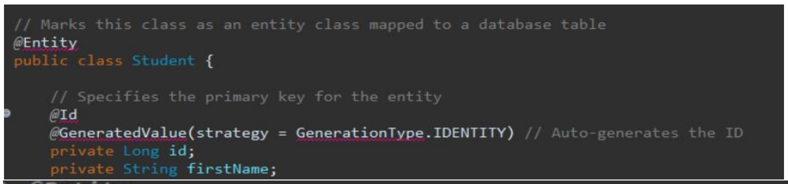
Why Do We Use `Model`?

- **Purpose:** The `Model` object is used to pass data from the controller to the view. It acts as a bridge, carrying information that the view needs to display.
- **Example Usage:** If you need to display a list of students on a webpage, you pass this list to the view using `model.addAttribute()`. The view (e.g., a Thymeleaf template) can then access the `students` attribute to display the data.

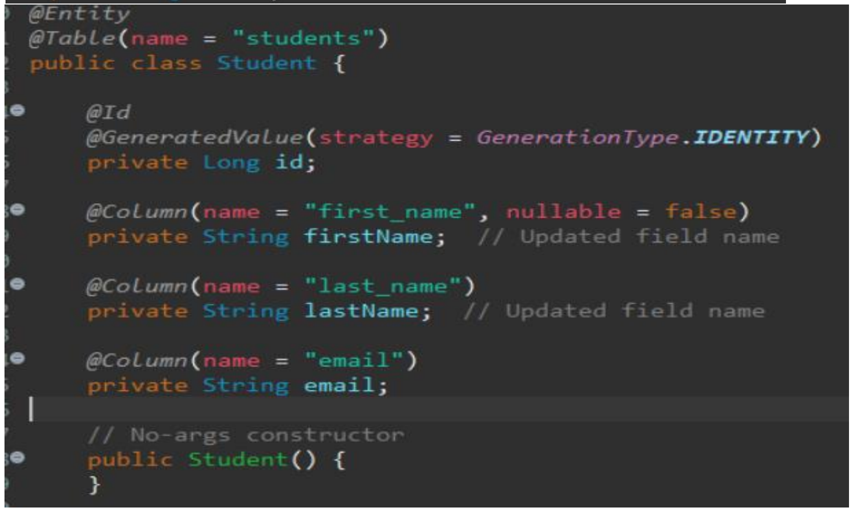
SpringBoot dev tool dependency →no need to re-run Apllication Application will automatically run after save

1. Student Entity Class

Definition: This is a simple Java class annotated with @Entity, representing the Student table in the database.



```java
// Marks this class as an entity class mapped to a database table
@Entity
public class Student {

    // Specifies the primary key for the entity
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generates the ID
    private Long id;
    private String firstName;
```

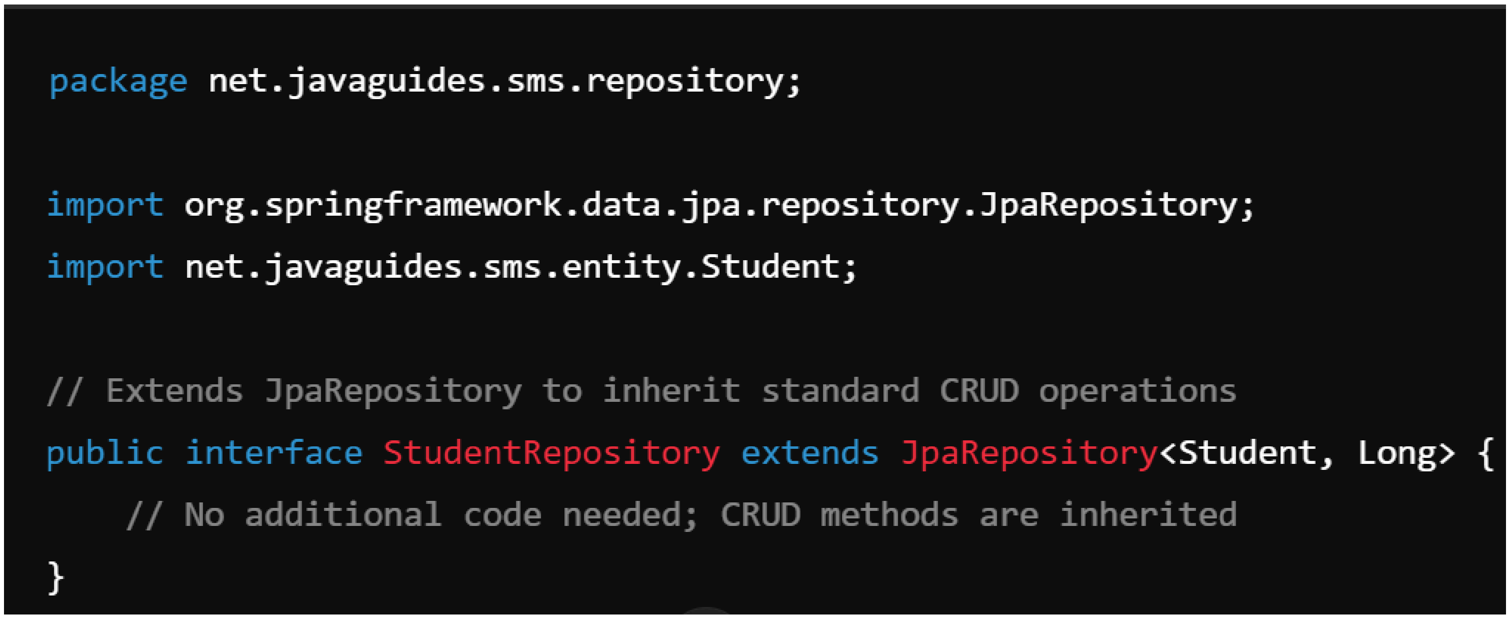


```java
@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name", nullable = false)
    private String firstName;  // Updated field name

    @Column(name = "last_name")
    private String lastName;  // Updated field name

    @Column(name = "email")
    private String email;

    // No-args constructor
    public Student() {
    }
```

---

2. StudentRepository Interface(Studententity.java)

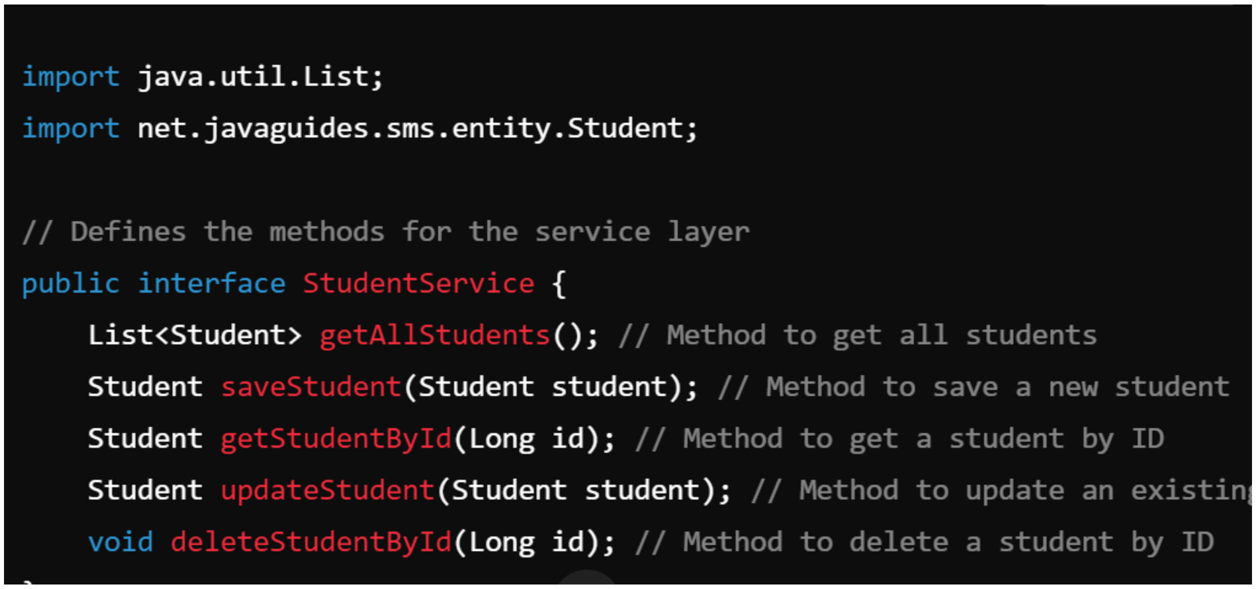
Definition: This is a repository interface extending JpaRepository to provide CRUD operations on the Student entity.



```java
package net.javaguides.sms.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import net.javaguides.sms.entity.Student;

// Extends JpaRepository to inherit standard CRUD operations
public interface StudentRepository extends JpaRepository<Student, Long> {
    // No additional code needed; CRUD methods are inherited
}
```

3. StudentService Interface

Definition: This is a service interface to define methods for handling business logic.

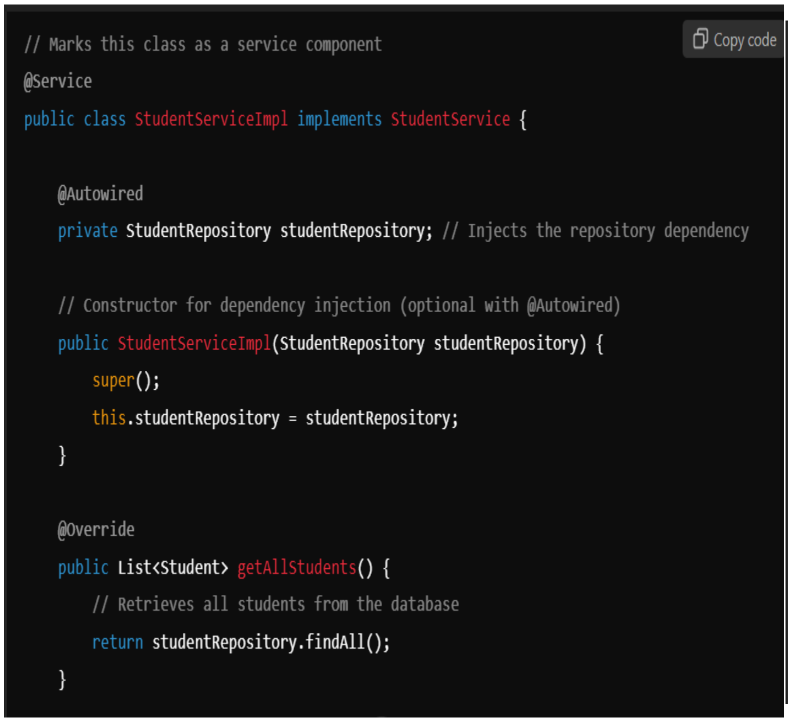


```java
import java.util.List;
import net.javaguides.sms.entity.Student;

// Defines the methods for the service layer
public interface StudentService {
    List<Student> getAllStudents(); // Method to get all students
    Student saveStudent(Student student); // Method to save a new student
    Student getStudentById(Long id); // Method to get a student by ID
    Student updateStudent(Student student); // Method to update an existin
    void deleteStudentById(Long id); // Method to delete a student by ID
```

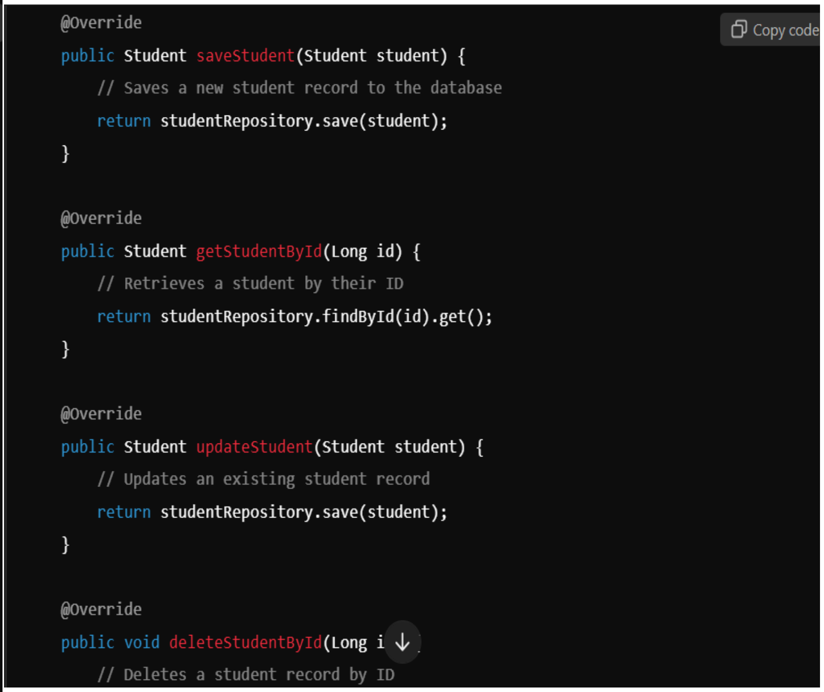
----

4. StudentServiceImpl Class

Definition: This class implements the StudentService interface and uses StudentRepository to perform database operations.



```java
// Marks this class as a service component
@Service
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentRepository studentRepository; // Injects the repository dependency

    // Constructor for dependency injection (optional with @Autowired)
    public StudentServiceImpl(StudentRepository studentRepository) {
        super();
        this.studentRepository = studentRepository;
    }

    @Override
    public List<Student> getAllStudents() {
        // Retrieves all students from the database
        return studentRepository.findAll();
    }
```

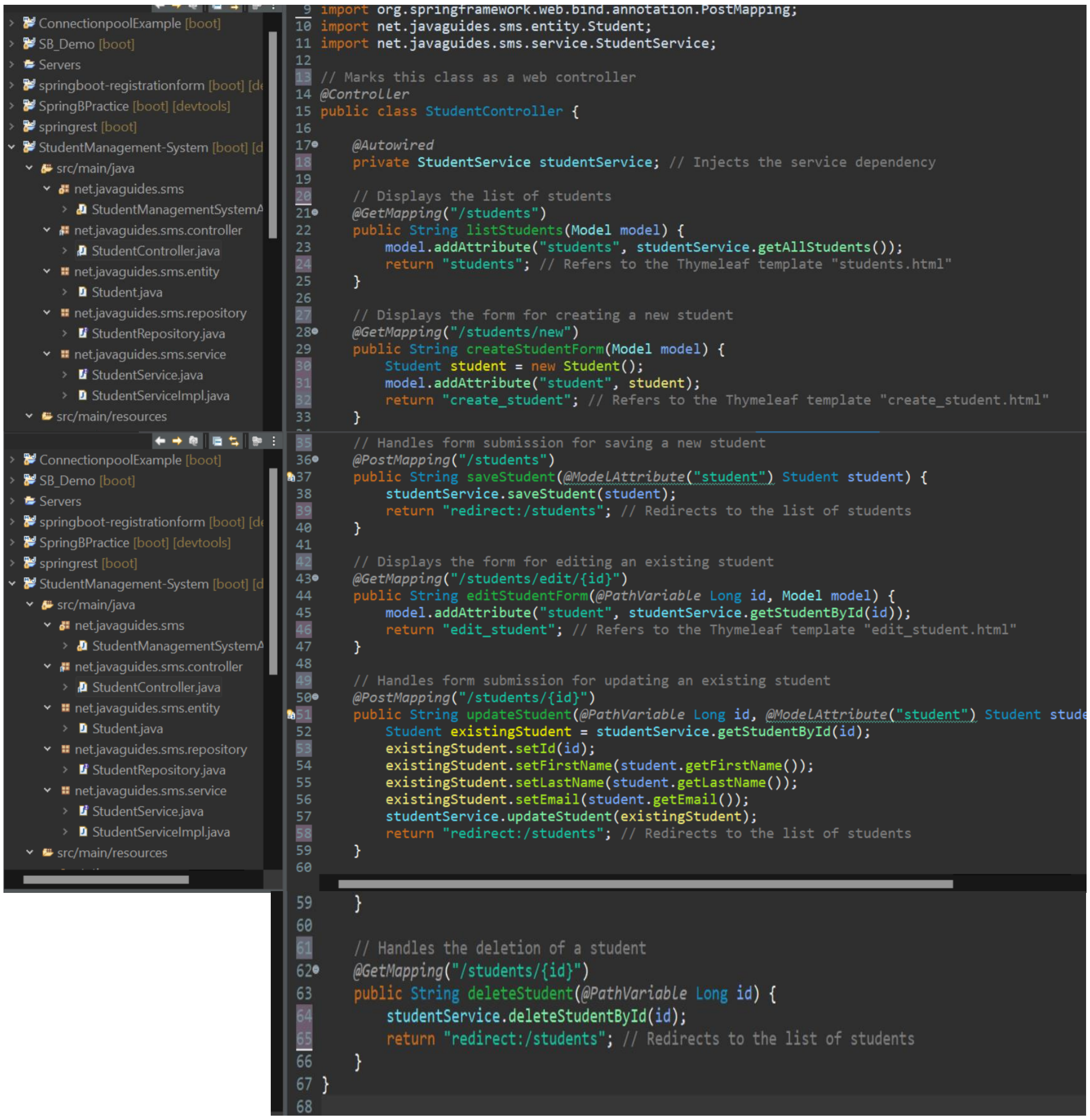


```java
@Override
public Student saveStudent(Student student) {
    // Saves a new student record to the database
    return studentRepository.save(student);
}

@Override
public Student getStudentById(Long id) {
    // Retrieves a student by their ID
    return studentRepository.findById(id).get();
}

@Override
public Student updateStudent(Student student) {
    // Updates an existing student record
    return studentRepository.save(student);
}

@Override
public void deleteStudentById(Long i
    // Deletes a student record by ID
```

----

5. StudentController Class

Definition: This is a Spring MVC controller that handles HTTP requests and interacts with the service layer.

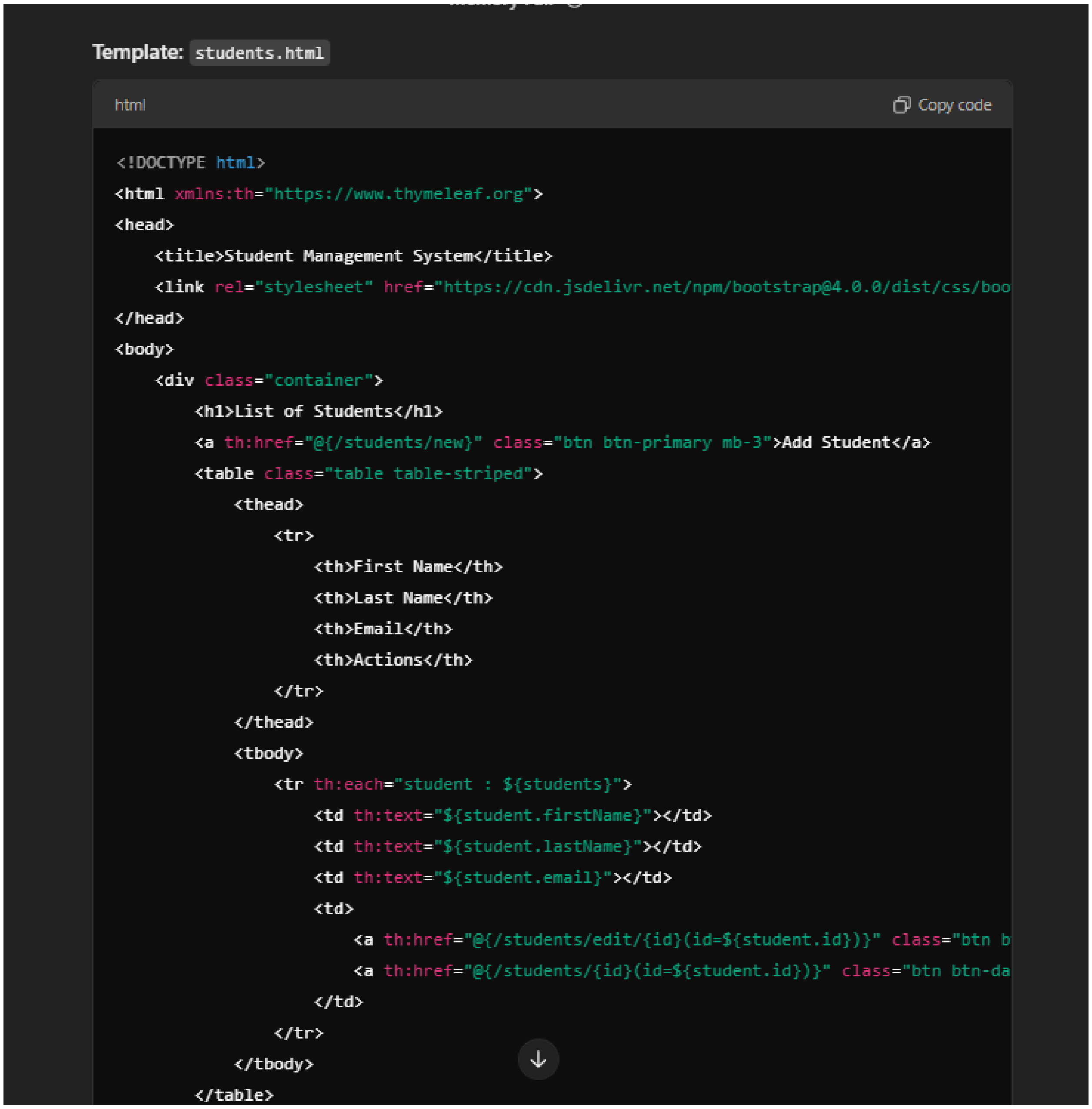

ConnectionpoolExample [boot]
SB_Demo [boot]
Servers
springboot-registrationform [boot] [de
SpringBPractice [boot] [devtools]
springrest [boot]
StudentManagement-System [boot] [d
src/main/java
net.javaguides.sms
StudentManagementSystemA
net.javaguides.sms.controller
StudentController.java
net.javaguides.sms.entity
Student.java
net.javaguides.sms.repository
StudentRepository.java
net.javaguides.sms.service
StudentService.java
StudentServiceImpl.java
src/main/resources

```java
import org.springframework.web.bind.annotation.PostMapping;
import net.javaguides.sms.entity.Student;
import net.javaguides.sms.service.StudentService;

// Marks this class as a web controller
@Controller
public class StudentController {

    @Autowired
    private StudentService studentService; // Injects the service dependency

    // Displays the list of students
    @GetMapping("/students")
    public String listStudents(Model model) {
        model.addAttribute("students", studentService.getAllStudents());
        return "students"; // Refers to the Thymeleaf template "students.html"
    }

    // Displays the form for creating a new student
    @GetMapping("/students/new")
    public String createStudentForm(Model model) {
        Student student = new Student();
        model.addAttribute("student", student);
        return "create_student"; // Refers to the Thymeleaf template "create_student.html"
    }
    // Handles form submission for saving a new student
    @PostMapping("/students")
    public String saveStudent(@ModelAttribute("student") Student student) {
        studentService.saveStudent(student);
        return "redirect:/students"; // Redirects to the list of students
    }

    // Displays the form for editing an existing student
    @GetMapping("/students/edit/{id}")
    public String editStudentForm(@PathVariable Long id, Model model) {
        model.addAttribute("student", studentService.getStudentById(id));
        return "edit_student"; // Refers to the Thymeleaf template "edit_student.html"
    }

    // Handles form submission for updating an existing student
    @PostMapping("/students/{id}")
    public String updateStudent(@PathVariable Long id, @ModelAttribute("student") Student stude
        Student existingStudent = studentService.getStudentById(id);
        existingStudent.setId(id);
        existingStudent.setFirstName(student.getFirstName());
        existingStudent.setLastName(student.getLastName());
        existingStudent.setEmail(student.getEmail());
        studentService.updateStudent(existingStudent);
        return "redirect:/students"; // Redirects to the list of students
    }

    // Handles the deletion of a student
    @GetMapping("/students/{id}")
    public String deleteStudent(@PathVariable Long id) {
        studentService.deleteStudentById(id);
        return "redirect:/students"; // Redirects to the list of students
    }
}
```

6. Thymeleaf Templates

Definition: These are the HTML files for displaying web content.

Template: students.html



Template: students.html

html Copy code

```html
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org">
<head>
    <title>Student Management System</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/boo
</head>
<body>
    <div class="container">
        <h1>List of Students</h1>
        <a th:href="@{/students/new}" class="btn btn-primary mb-3">Add Student</a>
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>First Name</th>
                    <th>Last Name</th>
                    <th>Email</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="student : ${students}">
                    <td th:text="${student.firstName}"></td>
                    <td th:text="${student.lastName}"></td>
                    <td th:text="${student.email}"></td>
                    <td>
                        <a th:href="@{/students/edit/{id}(id=${student.id})}" class="btn b
                        <a th:href="@{/students/{id}(id=${student.id})}" class="btn btn-da
                    </td>
                </tr>
            </tbody>
        </table>
```

Summary:

- **Entity**: Represents database objects.
- **Repository**: Provides CRUD operations.
- **Service**: Contains business logic.
- **Controller**: Handles HTTP requests and interacts with the service.
- **Templates**: Render the UI using Thymeleaf.

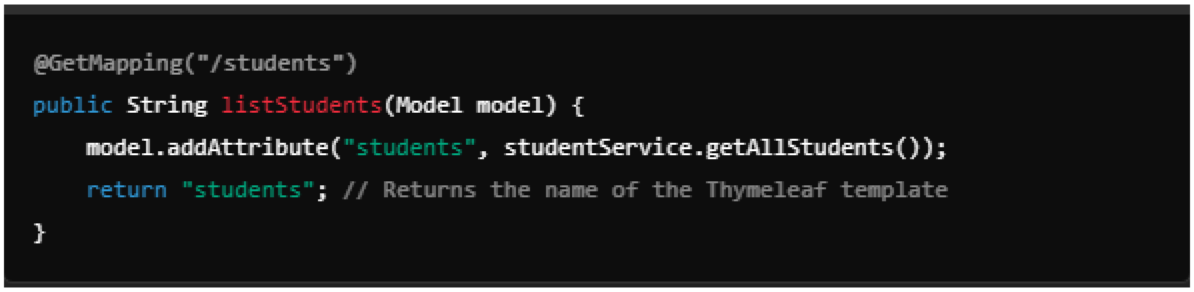
To understand the flow of the code from start to finish, let's outline the sequence of actions and interactions between the components of the Student Management System:

1. Client Request

- The process begins when a user interacts with the web application through the UI (e.g., visiting a URL or submitting a form).

2. Controller Layer (StudentController)

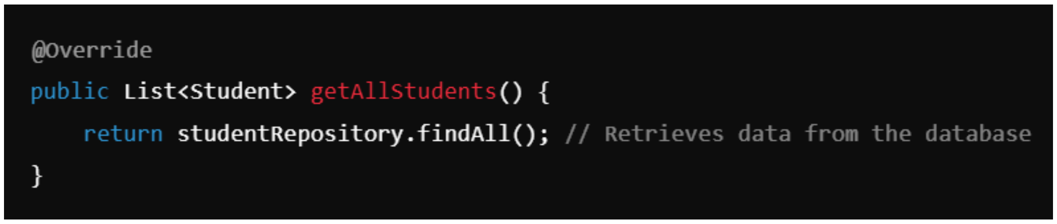
- The request is first handled by the StudentController, which contains various handler methods mapped to specific HTTP endpoints.
- Example: When a user visits /students, the listStudents method in StudentController is triggered.



```java
@GetMapping("/students")
public String listStudents(Model model) {
    model.addAttribute("students", studentService.getAllStudents());
    return "students"; // Returns the name of the Thymeleaf template
}
```

3. Service Layer (StudentService & StudentServiceImpl)

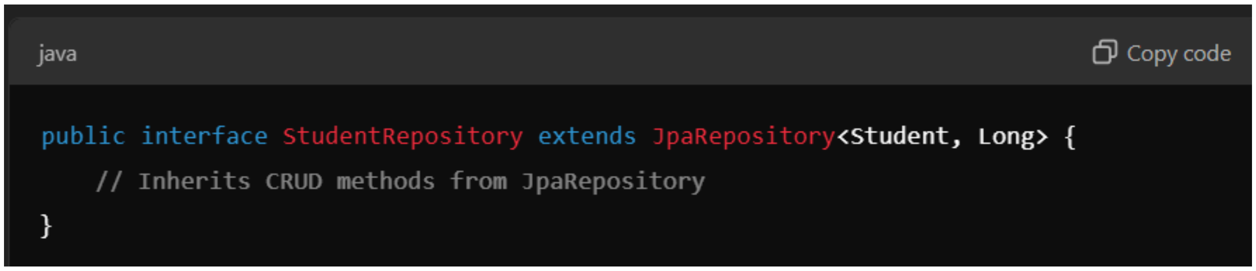
- The StudentController interacts with the StudentService interface to call the appropriate business logic methods.
- StudentServiceImpl, the implementation class, handles the actual logic by calling methods from StudentRepository.



```java
@Override
public List<Student> getAllStudents() {
    return studentRepository.findAll(); // Retrieves data from the database
}
```

4. Repository Layer (StudentRepository)

- The StudentServiceImpl class uses StudentRepository, which extends JpaRepository, to interact with the database.
- This repository layer abstracts common database operations, such as findAll(), save(), findById(), and deleteById().



```java
public interface StudentRepository extends JpaRepository<Student, Long> {
    // Inherits CRUD methods from JpaRepository
}
```

5. Database Interaction

- The StudentRepository uses Spring Data JPA to communicate with the database and perform the requested operations (e.g., fetching all student records).

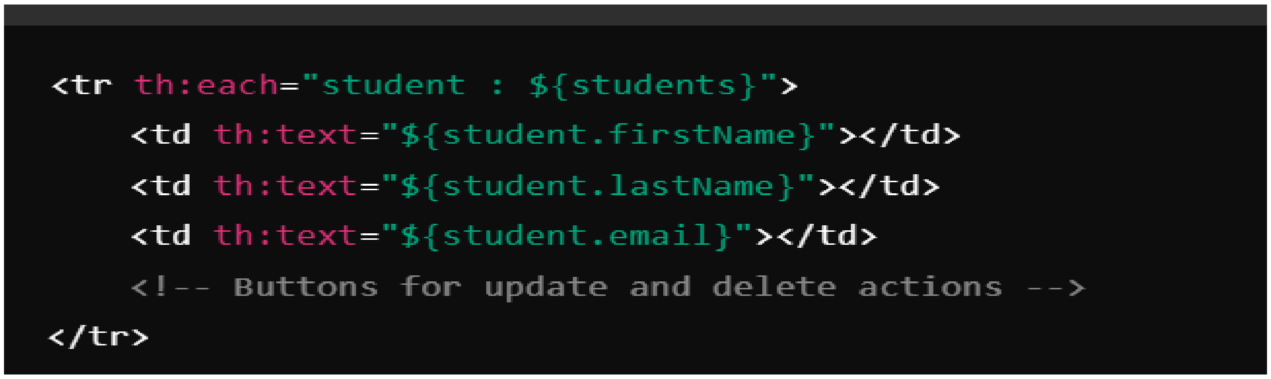
6. Response Handling

- Once the data is retrieved or modified by the StudentServiceImpl, it is passed back to the StudentController.
- The StudentController then sends this data to the appropriate Thymeleaf template.

model.addAttribute("students", studentService.getAllStudents()); // Adds data to the model

7. Thymeleaf View Rendering

- The students.html or any relevant template receives the data model from the controller.
- Thymeleaf processes the template and dynamically renders the HTML with the data.



```html
<tr th:each="student : ${students}">
    <td th:text="${student.firstName}"></td>
    <td th:text="${student.lastName}"></td>
    <td th:text="${student.email}"></td>
    <!-- Buttons for update and delete actions -->
</tr>
```

8. Response Sent to the Client

- The rendered HTML page is sent back to the client's browser, displaying the requested information or confirming an action (e.g., displaying all students or confirming that a student was added).

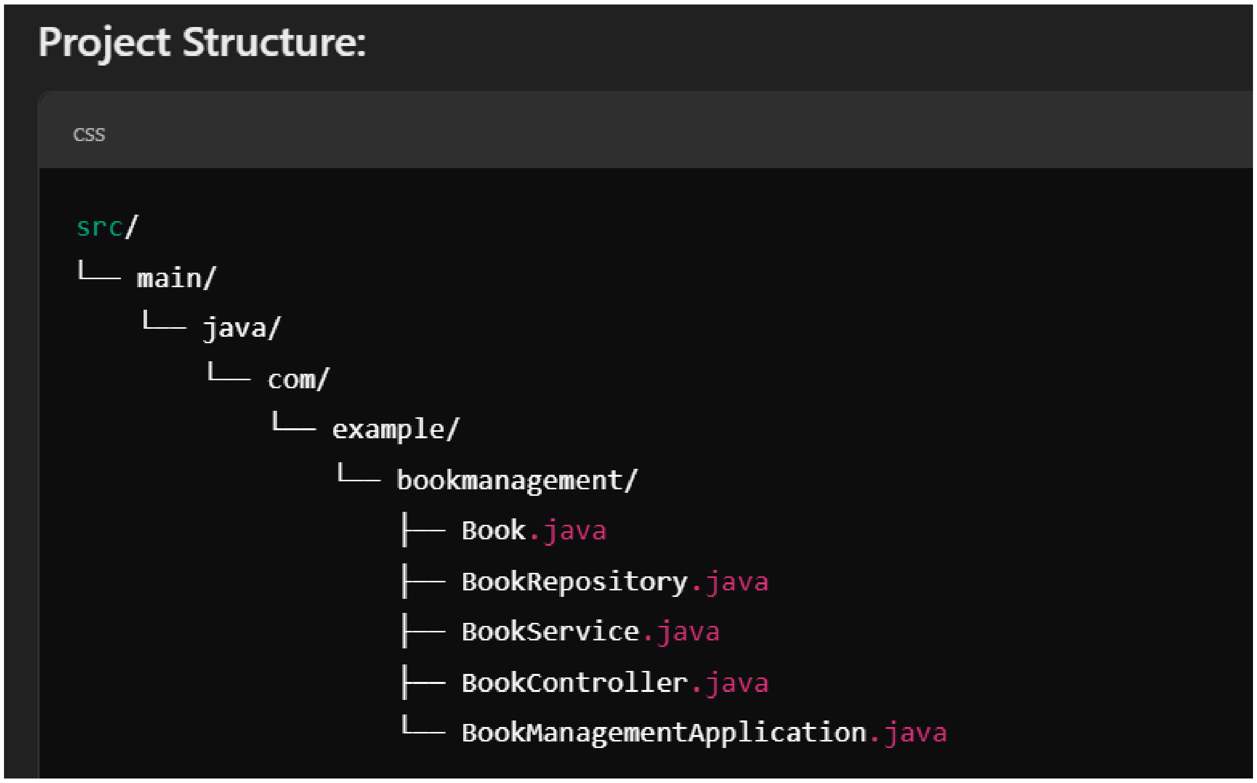
Typical User Flows

1. **View All Students**:
    - User visits /students.
    - StudentController calls StudentService.getAllStudents().
    - Data is fetched from StudentRepository and displayed in students.html.
2. **Add a New Student**:
    - User clicks "Add Student" (navigates to /students/new).
    - StudentController displays create_student.html.
    - User submits the form.
    - StudentController calls StudentService.saveStudent(student), saving it to the database.
    - Redirects to /students to show the updated list.
3. **Edit a Student**:
    - User clicks "Update" for a student (navigates to /students/edit/{id}).
    - StudentController fetches the student by ID using StudentService.getStudentById(id) and displays edit_student.html.
    - User modifies and submits the form.
    - StudentController calls StudentService.updateStudent(student).
    - Redirects to /students.
4. **Delete a Student**:
    - User clicks "Delete" for a student (navigates to /students/{id}).
    - StudentController calls StudentService.deleteStudentById(id), which uses StudentRepository.deleteById(id).
    - Redirects to /students.

Summary of Flow:

1. **Request** from client → **Controller** (handles HTTP request).
2. **Controller** → **Service** (handles business logic).
3. **Service** → **Repository** (interacts with the database).
4. **Repository** → **Database** (CRUD operations).
5. **Data** returns from **Repository** → **Service** → **Controller**.
6. **Controller** → **Thymeleaf Template** (renders view).
7. **Response** sent back to the **client**.

Spring Boot web Application

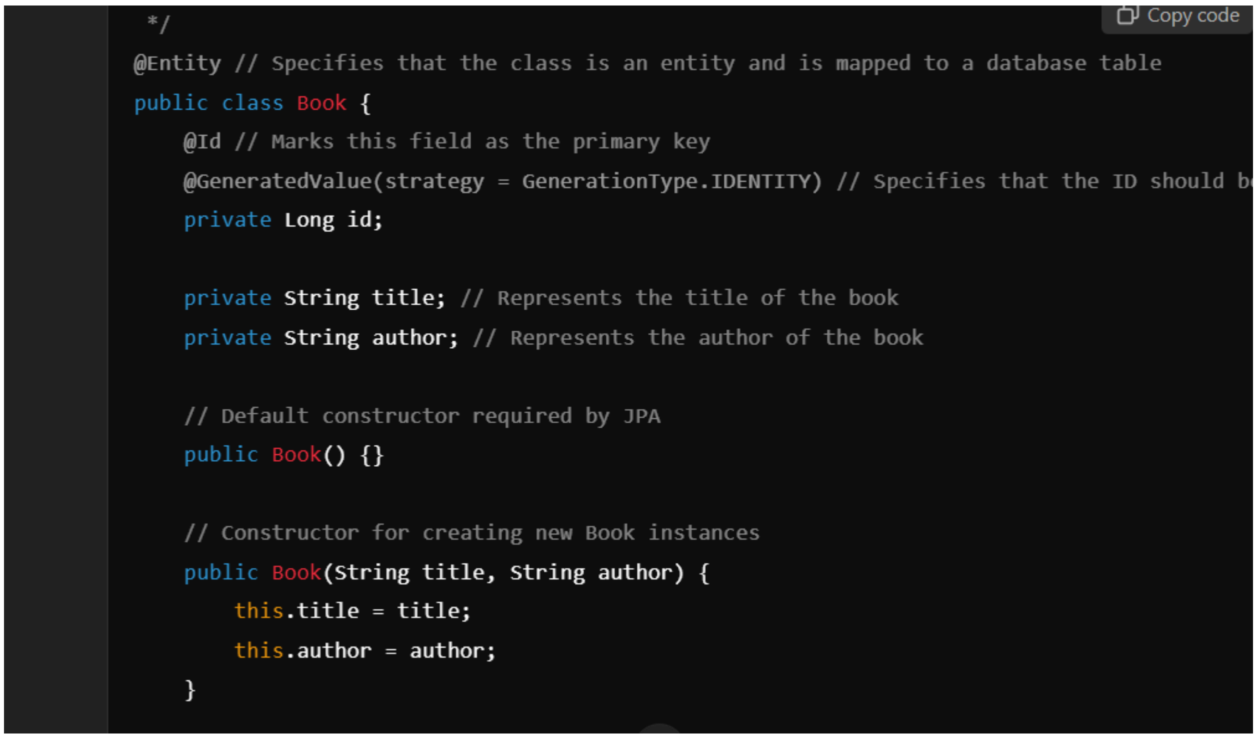


Project Structure:

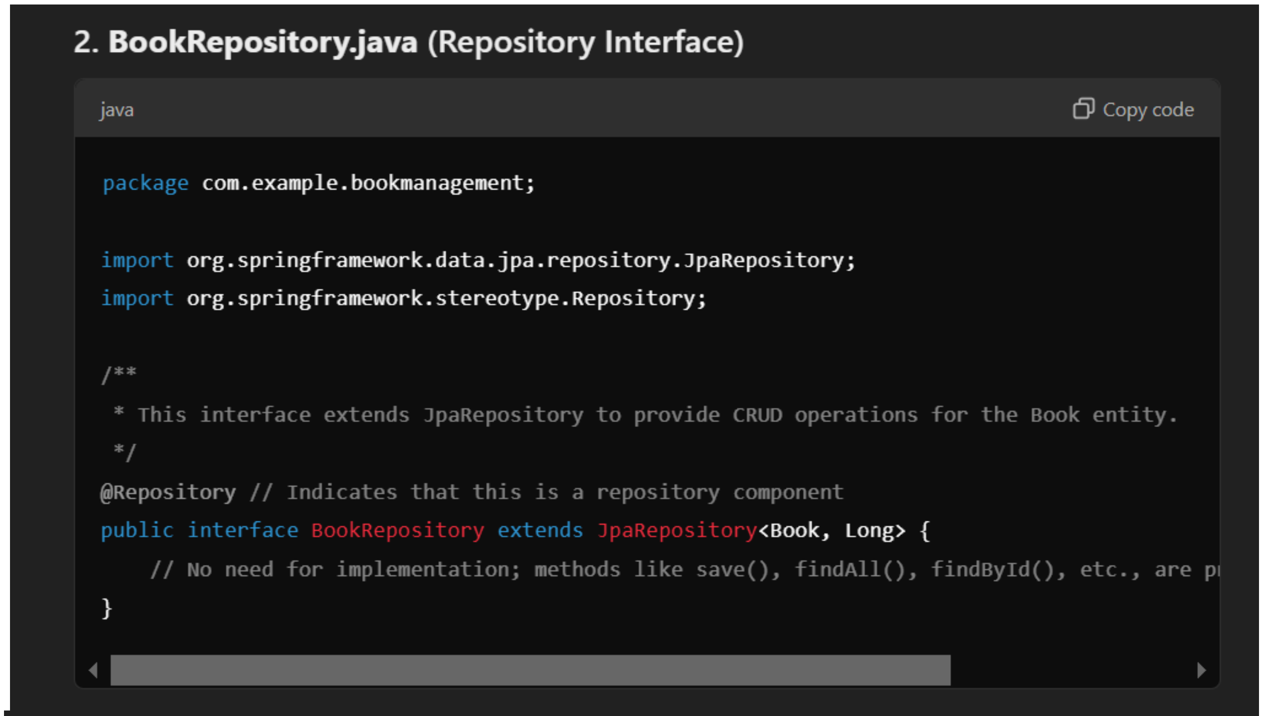
css

```
src/
└── main/
    └── java/
        └── com/
            └── example/
                └── bookmanagement/
                    ├── Book.java
                    ├── BookRepository.java
                    ├── BookService.java
                    ├── BookController.java
                    └── BookManagementApplication.java
```

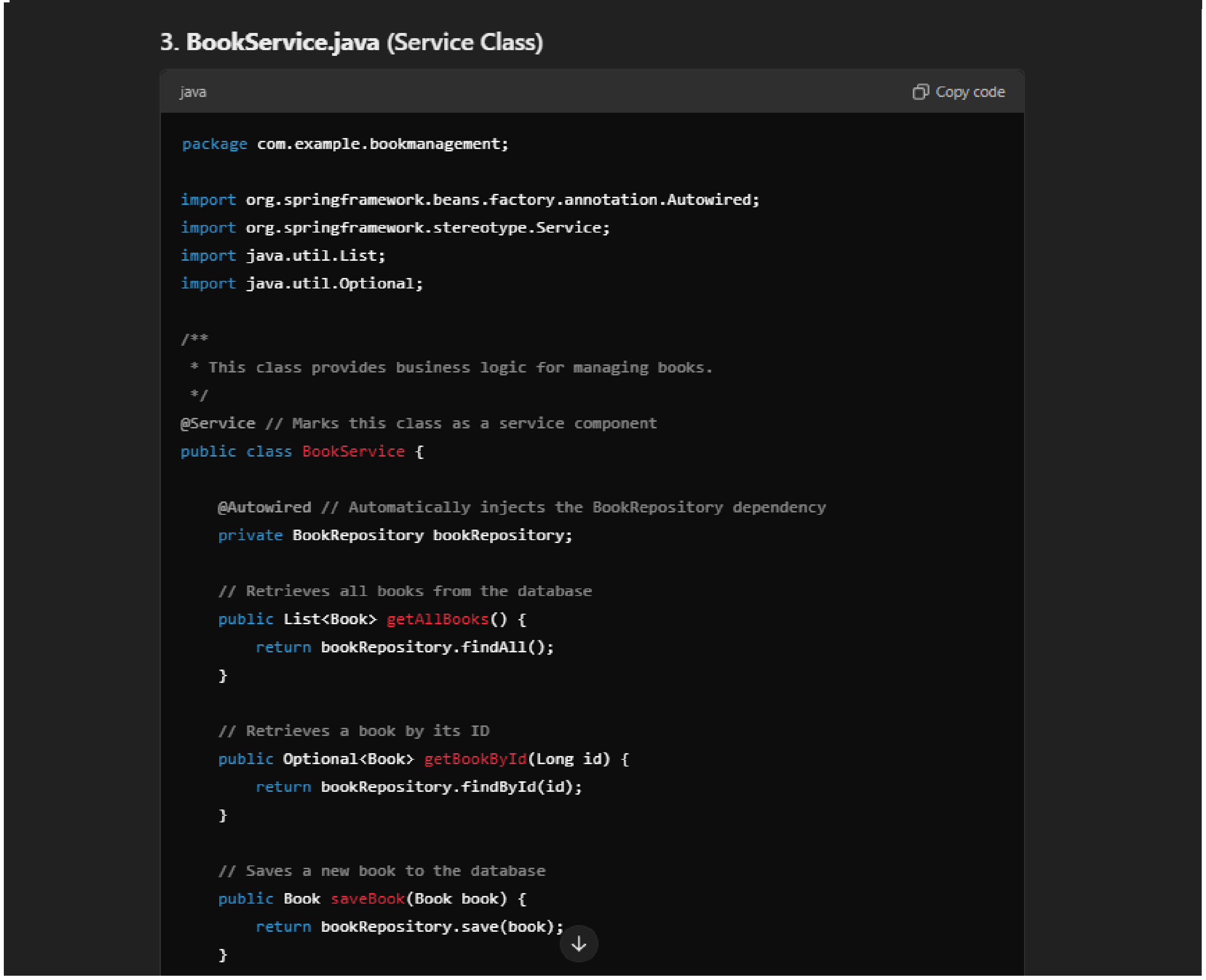
Book.java (Entity class)



```
 */
@Entity // Specifies that the class is an entity and is mapped to a database table
public class Book {
    @Id // Marks this field as the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Specifies that the ID should b
    private Long id;

    private String title; // Represents the title of the book
    private String author; // Represents the author of the book

    // Default constructor required by JPA
    public Book() {}

    // Constructor for creating new Book instances
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
```

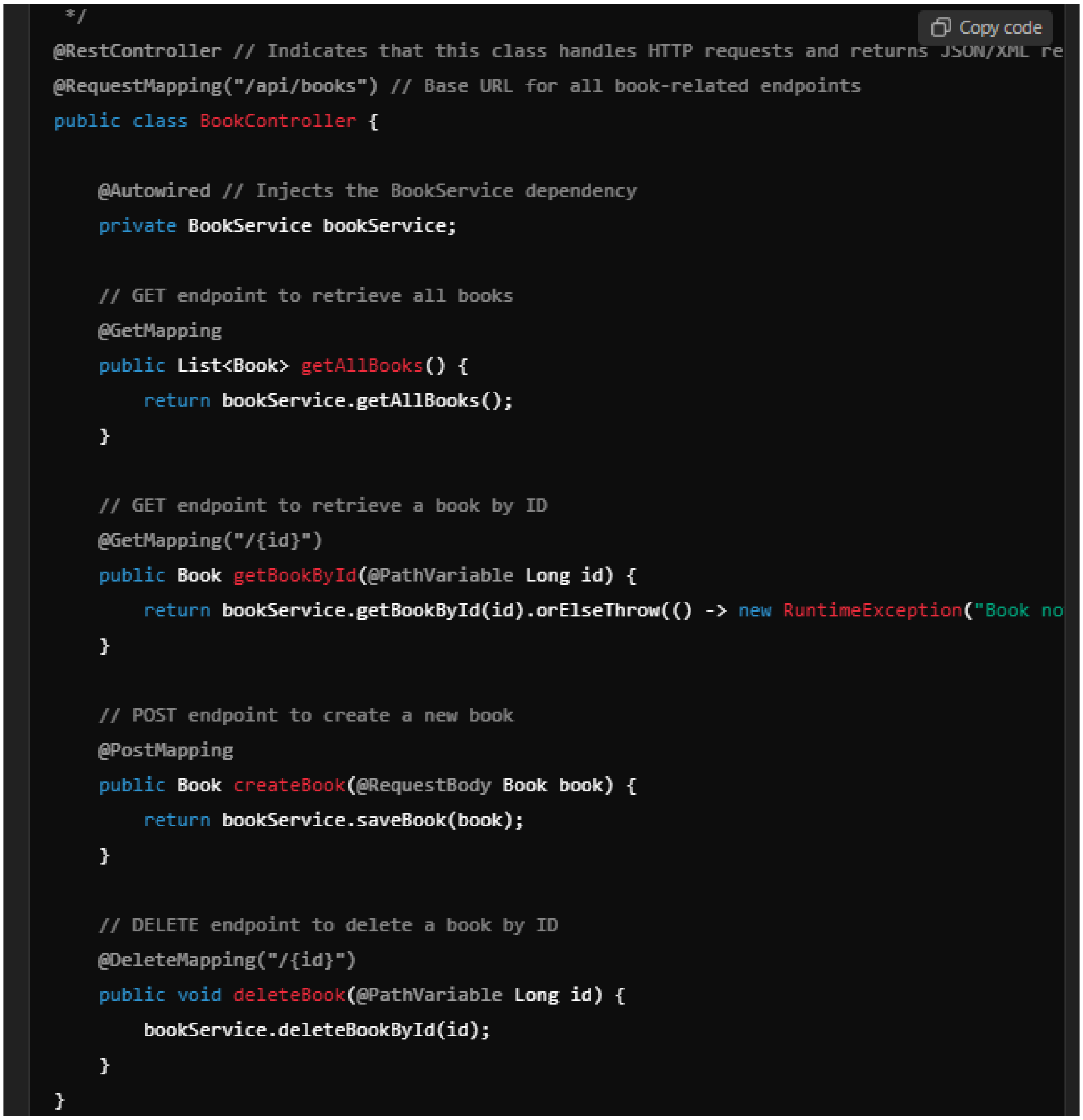

2. BookRepository.java (Repository Interface)

```java
package com.example.bookmanagement;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

/**
 * This interface extends JpaRepository to provide CRUD operations for the Book entity.
 */
@Repository // Indicates that this is a repository component
public interface BookRepository extends JpaRepository<Book, Long> {
    // No need for implementation; methods like save(), findAll(), findById(), etc., are p
}
```

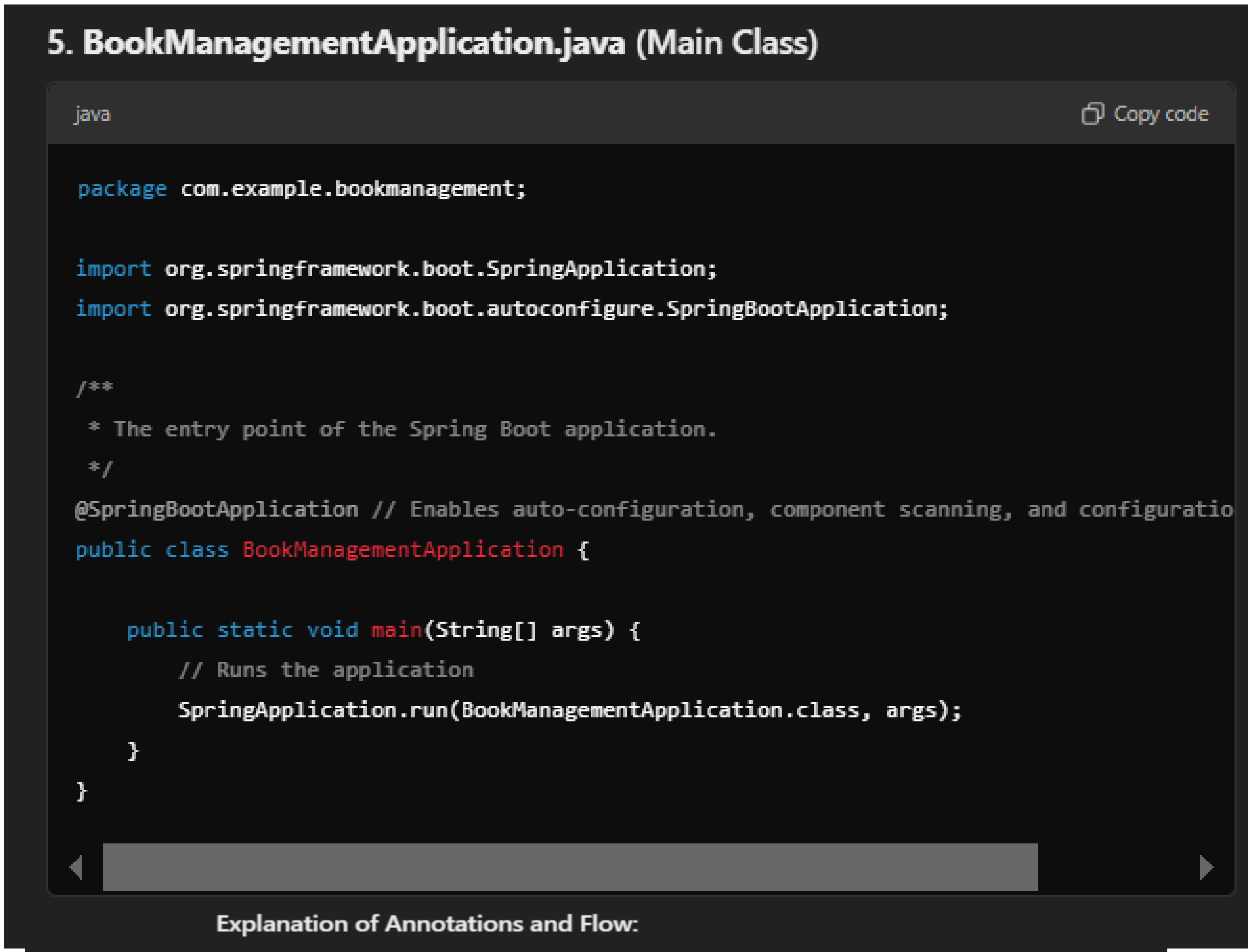

3. BookService.java (Service Class)

```java
package com.example.bookmanagement;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

/**
 * This class provides business logic for managing books.
 */
@Service // Marks this class as a service component
public class BookService {

    @Autowired // Automatically injects the BookRepository dependency
    private BookRepository bookRepository;

    // Retrieves all books from the database
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }

    // Retrieves a book by its ID
    public Optional<Book> getBookById(Long id) {
        return bookRepository.findById(id);
    }

    // Saves a new book to the database
    public Book saveBook(Book book) {
        return bookRepository.save(book);
    }
```

Controller.java

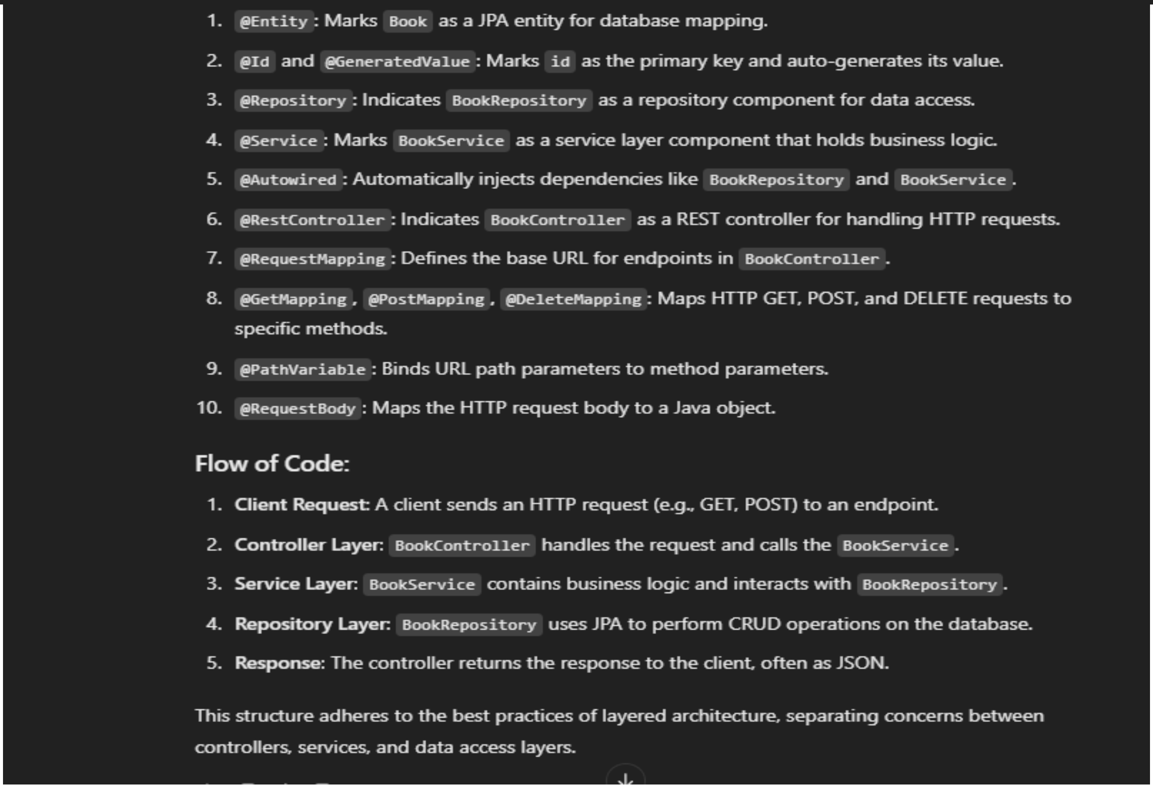


```java
 */
@RestController // Indicates that this class handles HTTP requests and returns JSON/XML re
@RequestMapping("/api/books") // Base URL for all book-related endpoints
public class BookController {

    @Autowired // Injects the BookService dependency
    private BookService bookService;

    // GET endpoint to retrieve all books
    @GetMapping
    public List<Book> getAllBooks() {
        return bookService.getAllBooks();
    }

    // GET endpoint to retrieve a book by ID
    @GetMapping("/{id}")
    public Book getBookById(@PathVariable Long id) {
        return bookService.getBookById(id).orElseThrow(() -> new RuntimeException("Book no
    }

    // POST endpoint to create a new book
    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookService.saveBook(book);
    }

    // DELETE endpoint to delete a book by ID
    @DeleteMapping("/{id}")
    public void deleteBook(@PathVariable Long id) {
        bookService.deleteBookById(id);
    }
}
```



5. BookManagementApplication.java (Main Class)

```java
package com.example.bookmanagement;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * The entry point of the Spring Boot application.
 */
@SpringBootApplication // Enables auto-configuration, component scanning, and configuratio
public class BookManagementApplication {

    public static void main(String[] args) {
        // Runs the application
        SpringApplication.run(BookManagementApplication.class, args);
    }
}
```



Explanation of Annotations and Flow:

1. `@Entity`: Marks `Book` as a JPA entity for database mapping.
2. `@Id` and `@GeneratedValue`: Marks `id` as the primary key and auto-generates its value.
3. `@Repository`: Indicates `BookRepository` as a repository component for data access.
4. `@Service`: Marks `BookService` as a service layer component that holds business logic.
5. `@Autowired`: Automatically injects dependencies like `BookRepository` and `BookService`.
6. `@RestController`: Indicates `BookController` as a REST controller for handling HTTP requests.
7. `@RequestMapping`: Defines the base URL for endpoints in `BookController`.
8. `@GetMapping`, `@PostMapping`, `@DeleteMapping`: Maps HTTP GET, POST, and DELETE requests to specific methods.
9. `@PathVariable`: Binds URL path parameters to method parameters.
10. `@RequestBody`: Maps the HTTP request body to a Java object.

Flow of Code:

1. **Client Request:** A client sends an HTTP request (e.g., GET, POST) to an endpoint.
2. **Controller Layer:** `BookController` handles the request and calls the `BookService`.
3. **Service Layer:** `BookService` contains business logic and interacts with `BookRepository`.
4. **Repository Layer:** `BookRepository` uses JPA to perform CRUD operations on the database.
5. **Response:** The controller returns the response to the client, often as JSON.

This structure adheres to the best practices of layered architecture, separating concerns between controllers, services, and data access layers.